JSON-Format für smarthome.py

Um smarthome.py-conf-Files in JSON editierbar zu machen, ist kein großer Lernaufwand nötig. Es gibt allerdings ein paar (Komfort-)Erweiterungen, die man lernen muss.

Grundkonzept

Die Grundregel ist ganz einfach:

- Jedes Item in einem conf-File mit der Notation
  `[[[[Schlafzimmer]]]]`
  wird in JSON zu einem Objekt:
  `"Schlafzimmer" : { .... }`
  Dabei muss der Item-Name mit einem Großbuchstaben beginnen. Nur Items, die mit einem Großbuchstaben beginnen, wird die Schema-Prüfung durchlaufen, kleingeschriebene Items sind möglich, geben keinen Fehler und werden nicht weiter geprüft (incl. aller Unteritems).

- Jede Item-Eigenschaft (Property) in einem conf-File mit der Notation
  `name = Schlafzimmer`
  wird zu einem JSON-Property:
  `"name": "Schlafzimmer"`
  Dabei muss der Property-Name mit einem Kleinbuchstaben beginnen. Auch hier gilt: Andere Notationen sind möglich, bei Properties werden aber nur bekannte Properties geprüft. Alle anderen werden nicht weiter geprüft, aber durchaus konvertiert. Dies ist notwendig, damit man plugin-Properties notieren kann, die im ItemSchema noch nicht enthalten sind.

Beispiel: Folgendes JSON

```
{
    "$schema": "../schema/ItemSchema.json",
    "Test": {
        "type": "bool",
        "name": "Nur ein Test",
        "knx_dpt": "1",
        "knx_send": "3/4/4",
        "enforce_updates": true
    }
}
```

wird zu:

```
[Test]
       type = bool
```

```
name = Nur ein Test
knx_dpt = 1
knx_send = 3/4/4
enforce_updates = True
```

Anmerkung: Auf Top-Level ist aus technischen Gründen ein Property "$schema" notwendig, das immer den gleichen Inhalt hat und nicht konvertiert wird.

Mit diesem Grundkonzept kommt man schon sehr weit, es gibt allerdings noch ein paar Komfort-Erweiterungen, die die Arbeit vereinfachen.

List-Properties

smarthome.py erlaubt die Notation von mehreren Werten für ein Property, diese Werte werden mittels '|' getrennt. Das JSON-Aqivalent einer solchen Notation ist ein Array von Werten:

Beispiel: Folgendes JSON

```
"knx_listen": [ "1/2/3", "1/2/4", "1/2/5" ],
```

wird zu

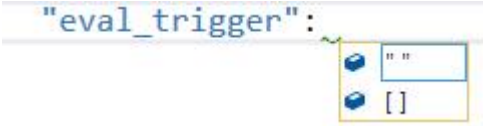
```
knx_listen = 1/2/3 | 1/2/4 | 1/2/5
```

Natürlich ist der Unterschied nichts sehr groß, der Vorteil von JSON ist aber, dass sich Arrays auch mehrzeilig notieren lassen

```
"knx_listen": [
    "1/2/3",
    "1/2/4",
    "1/2/5"
],
```

Wer schon mal viele Items bei eval_trigger eingetragen hat, wird die Übersichtlichkeit, die eine mehrzeilige Notation erlaubt, zu schätzen wissen.

Alle Properties, die im original-conf einen oder mehrere Werte zulassen, können im JSON als ein String (einfacher Wert) oder als Array notiert werden. Visual Studio zeigt dies in der code completion folgendermaßen an:



Falls ich hier irgendwelche Properties, die mehrere Werte zulassen, vergessen habe, bitte ne kurze Info an mich, werde ich im Schema und im Konverter ergänzen.

Mehrzeilige Properties

Einige Properties im conf-File können sehr lange Werte beinhalten. Dies ist vor allem beim sv_widget der Fall (was aber weiter unten abgehandelt wird). Für Properties mit sehr langen Werten ist es möglich, den Wert mehrzeilig zu notieren - als JSON-Array.

Dieses Feature ist derzeit für kein Property aktiviert, da ich noch keine praktische Notwendigkeit sah (ich aber on request jederzeit möglich).

Mehrzeilige List-Properties (derzeit nur sv_widget)

sv_widget ist ein Property, dass sehr lange Werte erlaubt UND gleichzeitig mehrere Werte pro Property. Damit muss sv_widget als Array von Arrays notiert werden. Hört sich kompliziert an, ist aber - sobald man es verstanden hat - sehr simpel.

Beispiel: Folgendes JSON

```
"sv_widget": [
    [
        "Wert 1, Teil 1 - ",
        "Wert 1, Teil 2 - ",
        "Wert 1, Teil 3"
    ],
    [
        "Wert 2, Teil 1 - ",
        "Wert 2, Teil 2"
    ],
    "Wert 3 - ohne weitere Teile"
]
```

wird zu

```
sv_widget = Wert 1, Teil 1 -Wert 1, Teil 2 -Wert 1, Teil 3 |
Wert 2, Teil 1 -Wert 2, Teil 2 | Wert 3 - ohne weitere Teile
```

(Den unteren Teil des Beispiels müsst ihr euch in einer Zeile denken).

Was übersichtlicher ist, muss jeder selbst entscheiden, natürlich kann man auch alles in einen String schreiben.

Wichtig ist: Das äußere Array beinhaltet die durch | verbundenen Werte, die ihrerseits ein inneres Array (oder einen einfachen String) haben können.

Objektartige Properties

Einige Properties im conf-File erlauben nur relativ kryptische Notation von

Werten, weil der Wert eigentlich aus mehreren Einzelwerten besteht (z.B. crontab). Als Alternative erlaubt der Konverter auch eine Notation als JSON-Objekt. Der Vorteil dieser Notation ist die Verifikation durch das Schema - die einzelnen Properties können auf zugelassene Wertebereiche geprüft und so Tippfehler vermieden werden. Grundsätzlich kann überall da, wo die Normale Notation erlaubt ist, auch die Objekt-Notation verwendet werden (bei crontab z.B. also auch im JSON-Array).

Im Folgenden werden diese Properties aufgelistet:

autotimer

Normale Notation	Objekt-Notation
`"100 = Wert"`	`{ "seconds": 100, "value": "Wert" }`
`"2m = Wert"`	`{ "minutes": 2, "value": "Wert" }`
`"80 = Wert"`	`{ "minutes": 1, "seconds": 20, "value": "Wert" }`

crontab

Normale Notation

```
"2 3 21 4 = Wert"
```

Objekt-Notation

```
{
    "minute": 2,
    "hour": 3,
    "day": 21,
    "weekday": "Friday",
    "value": "Wert"
}
```

```
"0 3 * * = Wert"
```

```
{
    "minute": 0,
    "hour": 3,
    "value": "Wert"
}
```

```
"0,15,30,45 3 * 5,6 = Wert"
```

```
{
    "minute": [0,15,30,45],
    "hour": 3,
    "weekday": ["Saturday", "Sunday"],
    "value": "Wert"
}
```

Weitere crontab-Angaben wie sunrise etc. werden weiterhin in der normalen Notation gemacht.

Benutzerspezifische Properties

Smarthome.py erlaubt es, in jedem Item beliebige benutzerspezifische Properties zu verwenden. Diese kann man im JSON-File genau so nutzen, indem man sie in der Form

```
"property": "wert"
```

notiert. Der Konverter konvertiert sie dann entsprechend ins conf-File

```
property = wert
```

Natürlich können benutzerspezifische Properties nicht über das JSON-Schema verifiziert werden - es gibt aber eine Möglichkeit, dieses um bentzerspezifische Properties zu erweitern. Dies ist weiter unten beschrieben.

Zusatz-Properties

Es gibt einige (wenige) Properties, die nur vom Konverter selbst ausgewertet werden und nicht für die conf-Files wichtig sind. Diese Properties beginnen alle mit einem "$"-Zeichen, um Kollisionen mit zukünftigen smarhome-Erweiterungen und Plugins zu vermeiden. Ausnahme ist hier "description", was aber historische Gründe hat.

Im folgenden sind die Properties aufgelistet:

description

Standard-JSON erlaubt keine Kommentare. Um trotzdem kommentarartige Anmerkungen im JSON notieren zu können, gibt es das Property "description". Es hat keine weitere Bedeutung und wird als Kommentar in das conf-File geschrieben.

In Visual Studio kann man im JSON kommentare benutzen, sowohl Zeilenkommentare, die mit // beginnen, als auch Block-Kommentare mit /* ... */.

Solche Kommentare können aber vom Konverter nicht in die .conf-Files überführt werden.

$schema

Ein Schemabasiertes JSON-Dokument sollte auf oberster Ebene ein "$schema"-Property haben, das das Schema angibt, gegen das dieses Dokument validiert werden kann. Dieses Property wird von Visual Studio Editor ausgewertet und wird deswegen in jedem File erwartet. Der Wert ist konstant, es hat immer die Form:

```
"$schema": "../schema/ItemSchema.json"
```

$order

Der JSON-Konverter übersetzt alle Items und Properties in der gleichen Reihenfolge, wie sie im JSON auftauchen, in das .conf-File. Durch die Angabe eines `"$order": Zahl` als Property eines Items kann man die Reihenfolge der Items verändern. Dazu werden alle Items auf gleicher Ebene anhand der Zahl, die hinter $order steht, aufsteigend sortiert.

Normalerweise braucht man das Feature nicht. Wenn man allerdings

Templates (s.u.) benutzt, wird die Reihenfolge der Items im .conf-File möglicherweise verändert (hängt mit dem Ersetzungsprozess beim Templating zusammen). Wenn man auf eine bestimmte Reihenfolge von Items angewiesen ist (dies ist z.B. im AutoBlind-Plugin wesentlich), kann man mit $order die gewünschte Reihenfolge im .conf-File erzeugen.

$delete, $replace, $template, $templated

Wird von der Template-Engine verwendet (s.u.) und auch in dem Zusammenhang erklärt.

Relative Adressierung von Items

Einige Properties im conf-File erlauben die Angabe von anderen Items. Zum Beispiel eval_trigger oder innerhalb von eval selbst. Der Konverter erlaubt neben der normalen (absoluten) Item-Adresse auch die Angabe einer relativen Adresse. Die Notation für relative Adressen basiert auf dem Zeichen '~' (Tilde). Die Syntax ist ganz einfach: Die erste '~' adressiert das aktuelle Item, jede weitere '~' dann eine weitere Stufe höher (Parent). Bei einem Item-Pfad

```
EG.EZ.Licht.Decke.Links
```

meint

```
~          EG.EZ.Licht.Decke.Links
~~         EG.EZ.Licht.Decke
~~~        EG.EZ.Licht
~~~~       EG.EZ
~~~~~      EG
```

wobei natürlich von da an immer mit der Punkt-Notation weiter referenziert werden kann

```
~~~.Wand   EG.EZ.Licht.Wand
```

Diese Referenzierung kann prinzipiell in jedem Property-Wert stehen. Auch im eval und sv_widget.

Bei eval gibt es noch eine kleine Vereinfachung (quasi Schreibkorrektur): Wenn in einem eval eine relative Adresse ohne 'sh.' gefunden wird, wird sh. automatisch davor gesetzt. Folgende Schreibweisen sind somit äquivalent:

```
"eval": "1 if sh.~~() else 0"  eval = 1 if sh.EG.EZ.Licht.Decke() else 0
"eval": "1 if ~~() else 0"     eval = 1 if sh.EG.EZ.Licht.Decke() else 0
```

Relative Adressen können nur innerhalb eines JSON-Files aufgelöst werden, es kommen Fehlermeldungen, wenn das Ziel-Item nicht im File gefunden

werden konnte.

Prüfung auf Existenz von Items

Der Konverter wurde mit dem Ziel geschrieben, möglichst viele Editier-Fehler im conf-File zu vermeiden. Ein wesentlicher Mehrwert wird dadurch erreicht, dass alle Item-Adressen auf Existenz geprüft werden. Wenn also irgendwo ein Item

```
~~~.Wand    EG.EZ.Licht.Wand
```

referenziert wird (egal ob es absolut oder relativ adressiert wurde), dann wird nach der Konvertierung aller Files geschaut, ob es dieses Item wirklich gibt. Falls nicht, wird eine Meldung ausgegeben, die angibt, welches Item referenziert werden sollte und in welcher Property die Referenz steht. Bei relativen Adressen stehen da 2 Meldungen, einmal mit der relativen Adresse und einmal mit der augelösten, da der Konverter nicht wissen kann, ob die relative oder die aufgelöste Adresse falsch ist.

Templates

Immer wieder verwendete Items mit Unteritems können in einer eigenen JSON-Datei abgelegt werden und als Template in anderen Dateien referenziert werden. Dabei können knx-Adressen übersetzt werden, so dass das gleiche Template an den verschiedenen Stellen, an den es verwendet wird, unterschiedliche knx-Adresen haben kann.

Templates sind experimentell, hier könnte es noch größere Änderungen geben, wobei ich bemüht bin, dass es keine zu großen Umstellungsaufwände gibt.

Warum sollte man Templates verwenden? Hier einige Vorteile:

- Änderungs-Komfort
  Man muss nur an einer Stelle etwas ändern, die Änderung wirkt aber an allen Stellen, an den das Template verwendet wird.

- Übersicht
  Die eigentlichen Item-Dateien werden kleiner und damit übersichtlicher. Man denkt eher in "Komponenten" wie RTR, Lüftung, Licht, Rolladen und nicht in einzelnen Items.

- Gleichheit
  Sobald ein Template funktioniert und getestet ist, funktioniert es an allen Stellen gleich. Wenn man mit Cut&Paste arbeitet, kann es schnell passieren, dass man bestimmte Files oder bestimmte Stellen vergisst - dann ist man wieder mal mit der Fehlersuche beschäftigt.

Beim Template kann das nicht passieren.

Der Hauptnachteil von Templates besteht darin, dass man im aktuellen JSON-File nur den Template-Namen und nicht dessen Ihnalt sieht. Bei einer sinnvollen Benennung der Templates ist dies aber gut zu beherrschen.

Ein Tempate ist ein normales JSON-File mit Items und deren Properties, das im templates-Verzeichnis liegt. In einem Template dürfen die knx-Gruppenadressen nur relativ formuliert werden, in der Form

```
"knx_listen": "+0/+1/+10"
```

Templates bekommen an der Stelle, wo sie eingefügt werden sollen, eine knx-Basisadresse, zu der die relativen Werte im Template addiert werden. Das Ergebnis der Addition ist dann die endgültige knx-Adresse für dieses Item

```
"$template": {"source": "test", "knx": "2/5/50"}
```

ergibt für obiges

```
knx_listen = 2/6/60
```

Als "shortcut" für +0 ist im Template auch ein '-' (Minuszeichen) erlaubt.

Achtung: Es ist auch möglich, in einem Template eine absolute Adresse (also onhe +) zu formulieren. Diese bleibt dann erhalten, die Basisadresse wird dann nicht angewendet. Das kann man nutzen, um sich in Templates auf Adressen zu beziehen, die sich bei verschiedenen Template-Nutzungen nicht ändern.

Achtung: Man kann bei der KNX-Adresse im Template auch gemischt relative und absolute Notation verwenden, z.B.

```
"knx_listen": "+0/7/+10"
```

Da die Adressen komponentenweise verrechnet werden (erst die Hauptgruppe, dann Mittelgruppe und dann Untergruppt), kann man bei passender GA-Struktur komfortabel verschiedene Adressierungsverhalten im Template realisieren. Obiges Beispiel führt zu

```
knx_listen = 2/7/60
```

Ich glaube, Templates versteht man am einfachsten anhand eines Beispiels:

template/rtr.json

```
{
    "$schema": "../schema/ItemSchema.json",
    "RTR": {
        "Ist": {
```

```
            "name": "Isttemperatur",
            "type": "num",
            "value": 21,
            "knx_dpt": "9",
            "knx_cache": "-/-/-",
            "sqlite": "yes"
        },
        "Soll": {
            "name": "Solltemperatur",
            "type": "num",
            "value": 20.5,
            "visu_acl": "rw",
            "sqlite": "yes",
            "knx_dpt": "9",
            "knx_cache": "-/-/+1",
            "knx_send": "-/-/+1",
            "crontab": [
                "0 6 * * = 22",
                "0 20 * * = 18"
            ]
        },
        "Stellwert": {
            "type": "num",
            "visu_acl": "ro",
            "sqlite": "yes",
            "knx_dpt": "5.001",
            "knx_cache": "-/-/+2"
        }
    }
}
```

Das ist ein stark vereinfachter RTR, der in vielen Räumen vorkommen könnte.

$template

Dieses Objekt hat genau 2 Properties:

- source
  Hier gibt man den Namen des Template-Files an, wobei ein File mit dem Name <name>.json im smarthome/templates-Verzeichnis stehen muss, bei source aber nur <name> angegeben wird.

- knx
  Dies ist die Basisadresse, zu der alle relativen Adressen im Template hinzugerechnet werden.

Das obige Template kann man jetzt in einem weiteren JSON-File verwenden.

```
items/test.json
{
```

```
    "$schema": "../schema/ItemSchema.json",
    "Test": {
        "$template": {"source": "rtr", "knx": "2/5/50"}
    }
}
```

Das Ergebnis sieht so aus:

```
items/test.conf
[Test]
       [[Temp]]
              name = Temperatur
              type = num
              value = 21
              knx_dpt = 9
              knx_cache = 2/5/50
              sqlite = yes
       [[Soll]]
              name = Solltemperatur
              type = num
              value = 20.50
              visu_acl = rw
              sqlite = yes
              knx_dpt = 9
              knx_cache = 2/5/51
              knx_send = 2/5/51
              crontab = 0 6 * * = 22 | 0 20 * * = 18
       [[Stellwert]]
              type = num
              visu_acl = ro
              sqlite = yes
              knx_dpt = 5.001
              knx_cache = 2/5/52
```

Das Ergebnis ist (hoffentlich) erwartungskonform. Wichtig: Der Name des obersten Items im Template (hier RTR) ist nicht relevant, beim Auflösen des Templates wird der Inhalt dieses Items (also alle Properties und Unteritems) in das Item transferiert, in dem die $template-deklaration steht (hier Test).

Operationen auf Templates

Beim Verwenden von Templates stellt man sehr schnell fest, dass es häufig Stellen gibt, an denen ein Template passen würde, allerdings braucht man irgendetwas spezielles, nur an dieser Stelle, man will z.B. bestimmte Properties oder Items anders haben (name im Template ist "Licht", an der verwendeten Stelle soll name aber "Deckenlicht" sein). Wäre blöd, wenn man deswegen das Template nicht verwenden könnte. Das hier vorgestellte Template-Konzept erlaubt auf Templates die Operationen:

- Überschreiben von Properties
- Ergänzen von Properties oder Items
- Löschen von Properties oder Items
- Ergänzen von Listen (JSON-Arrays)

Hier ein Beispiel:

```
{
    "$schema": "../schema/ItemSchema.json",
    "Test": {
        "$template": { "source": "rtr", "knx": "2/5/50" },
        "Temp": {
            "type": "num",
            "name": "Raumtemperatur",
            "enforce_updates": true,
            "Valid": {
                "description": "Ist die Temperatur noch gueltig",
                "name": "Gueltig?",
                "type": "bool",
                "eval": "1 if int(value) < 300 else 0",
                "eval_trigger": "~~",
                "autotimer": { "minutes": 6, "value": 500 }
            }
        },
        "Soll": {
            "$delete": [ "value" ],
            "type": "num",
            "crontab": [
                "0 18 * * = 20"
            ]
        },
        "$delete": ["Stellwert"]
    }
}
```

Folgende Änderungen werden hier gegenüber dem Template vorgenommen:

- Temp.type wird überschrieben (auch wenn es den gleichen Typ bekommen hat - soll nur zeigen, dass man das machen kann)
- Temp.name ist jetz Raumtemperatur statt Temperatur (Überschreiben von Properties)
- Temp.enforce_uptdates ist hinzugekommen (Ergänzen von Properties)
- Temp.Valid ist hinzugekommen (Ergänzen von Items)
- Soll.value ist gelöscht worden (Löschen von Properties)
- Soll.crontab hat jetzt zusätzlich einen weiteren Eintrag (Ergänzen von

Listen)

- Stellwert ist gelöscht worden (Löschen von Items)

Der Vollständigkeit halber hier noch das Ergebnis:

```
items/test.conf
[Test]
	[[Temp]]
		type = num
		name = Raumtemperatur
		enforce_updates = True
		value = 21
		knx_dpt = 9
		knx_cache = 2/5/50
		sqlite = yes
		[[[Valid]]]
			name = Gueltig?
			type = bool
			eval = 1 if int(value) < 300 else 0
			eval_trigger = Test.Temp
			autotimer = 6m = 500
	[[Soll]]
		type = num	//<-- value ist nicht mehr da
		crontab = 0 6 * * = 22 | 0 20 * * = 18 | 0 18 * * = 20
		name = Solltemperatur
		visu_acl = rw
		sqlite = yes
		knx_dpt = 9
		knx_cache = 2/5/51
		knx_send = 2/5/51
	//[[Sollwert]] <-- ist nicht mehr da
```

$delete

Das delete-Property wird nur von der Template-Engine ausgewertet und wird dazu verwendet, Properties oder ganze Items aus einem Template NICHT in das Zielobjekt zu übernehmen. Hinter delete gibt man eine Liste von Property-/Item-Namen an, die nicht übrenommen werden sollen.

Das Ganze ist nicht perfekt, derzeit lassen sich noch keine einzelnen Einträge von Listen löschen oder überschreiben - da arbeite ich noch dran.

$replace

Da man derzeit Listen nur Ergänzen kann oder mit $delete nur komplett löschen kann (und dann auch nicht neu anlegen kann, da das $delete nach dem Ganzen Ersetzungsprozess die Liste löscht), habe ich für den Fall, dass man eine Liste, die in einem Template steht, im aufrufenden Item neu machen will, $replace geschaffen. Das ist experimentell und wird zukünftig durch $delete abgelöst werden. Es wäre nett, wenn sich die Leute, die das

brauchen, bei mir mit Beispielen melden würden, dann könnte ich das mal "richtig" machen.

$templated

Dieses Property hat gar keine Auswirkung im Konverter, es wirkt nur auf die Prüfungen, die vom JSON-Schema durchgeführt werden.

Im JSON-Schema ist nicht nur festgelegt, wie die Properties geschrieben werden (für code completion) und welche Werte sie enthalten dürfen, sondern auch Abhängigkeiten, z.B. wenn knx_listen angegeben ist, muss auch knx_dpt angegeben sein.

Wenn man nun mit Templates arbeitet, kann es vorkommen, dass knx_dpt im Template steht, knx_listen aber in dem Item, das das Template verwendet. Nach der Template-Auflösung ist also das Item korrekt, aber zum Editierzeitpunkt wird ein Fehler angezeigt, da knx_listen ohne knx_dpt nicht ausreichend ist. Um diesen Fehler zu unterdürcken, kann man isTempated = true setzen, dadurch wird die Prüfung der Abhängigkeiten verhindert.

Templates in Templates

Templates können ihrerseits wiederum Templates referenzieren. Dadurch kann man Gleichteile von Templates in Unter-Templates rausziehen, was mehr Freiheiten beim Organisieren von Templates gibt.

Einziger Unterschied bei der Referenz eines Templates aus einem Template heraus ist, dass man keine knx-Basisadresse mitgeben darf.

Beispiel:

templates/temp.json

```
{
    "Temp": {
        "name": "Temperatur",
        "type": "num",
        "value": 21,
        "enforce_updates": true,
        "knx_dpt": "9",
        "knx_cache": "-/-/-",
        "sqlite": "yes",
        "Valid": {
            "description": "Ist die Temperatur noch gültig",
            "name": "Gültig?",
            "type": "bool",
            "eval": "1 if int(value) < 300 else 0",
            "eval_trigger": "~~",
            "autotimer": { "minutes": 6, "value": 500 }
        }
    }
```

```
}
```

templates/rtr.json

```
{
    "$schema": "../schema/ItemSchema.json",
    "RTR": {
        "Temp": {
            "$template": { "source": "temp" }
        },
        "Soll": {
            "name": "Solltemperatur",
            "type": "num",
            "value": 20.5,
            "visu_acl": "rw",
            "sqlite": "yes",
            "knx_dpt": "9",
            "knx_cache": "-/-/+1",
            "knx_send": "-/-/+1",
            "crontab": [
                { "minute": 0, "hour": 6, "value": 22 },
                { "minute": 0, "hour": 20, "value": 10 }
            ]
        },
        "Stellwert": {
            "type": "num",
            "visu_acl": "ro",
            "sqlite": "yes",
            "knx_dpt": "5.001",
            "knx_cache": "-/-/+2"
        }
    }
}
```

items/test.json

```
{
    "$schema": "../schema/ItemSchema.json",
    "Test": {
        "$template": { "source": "rtr", "knx": "2/5/50" },
        "Temp": {
            "name": "Raumtemperatur"
        },
        "Soll": {
            "description": "Das ist ein Test-Kommentar",
            "delete": [ "value" ],
            "type": "num",
            "crontab": [
                {
                    "minute": [ 15, 45 ],
                    "hour": [ 15, 18 ],
                    "weekday": [ "Monday", "Wednesday", "Friday" ],
                    "value": 20
                }
            ]
```

```
        },
        "delete": [ "Stellwert" ]
    }
}
```

An dem Beispiel sieht man, dass test.json das Template rtr.json verwendet, welches seinerseits für das Item "Temp" das Template temp.json nutzt.Zum Abschluss hier noch das Ergebnis der Konvertierung:

```
[Test]
       [[Temp]]
              name = Raumtemperatur
              type = num
              value = 21
              enforce_updates = True
              knx_dpt = 9
              knx_cache = 2/5/50
              sqlite = yes
              [[[Valid]]]
                     #Ist die Temperatur noch gültig
                     name = Gültig?
                     type = bool
                     eval = 1 if int(value) < 300 else 0
                     eval_trigger = Test.Temp
                     autotimer = 6m = 500
       [[Soll]]
              #Das ist ein Test-Kommentar
              type = num
              crontab = 0 6 * * = 22 | 0 20 * * = 10 | 15,45 15,18 * 0,2,4 = 20
              name = Solltemperatur
              visu_acl = rw
              sqlite = yes
              knx_dpt = 9
              knx_cache = 2/5/51
              knx_send = 2/5/51
```

GA Verifikation

Der Konfigurator gibt GA-Listen aus, die einen Abgleich mit der ETS leichter machen. Sie Listen werden in smarthome/items/GA geschrieben. Es sind bis zu 4 Listen möglich:

- Smarthome.txt
  Liste aller GA, die in den .conf-Files verwendet werden und der Items, in den sie verwendet werden.

- ETS.txt
  Liste aller GA, die in der ETS verwendet werden und der beschreibenden Texte

- FehlenInETS.txt

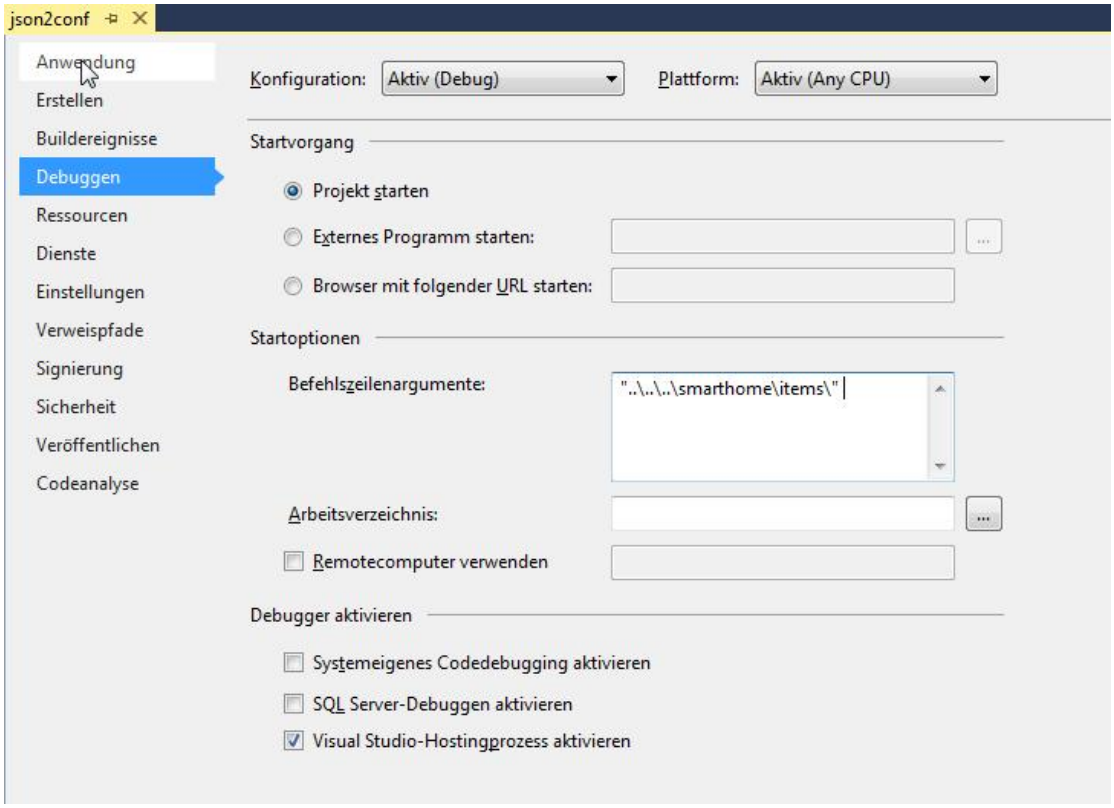
Liste aller GA, die in den .conf-Files vorkommen aber in der ETS fehlen

- FehlenInSmarthome.txt
  Liste aller GA, die in der ETS vorkommen, aber in den .conf-Files fehlen

Die erste Liste wird nach jedem Konverter-Lauf erzeugt. Für die anderen 3 Listen muss dem Konverter eine Referenz auf ein *.knxproj-File mitgegeben werden (ein Projekt-Export aus der ETS4). Um die Referenz im Projekt-Setup einzutragen, muss folgendermaßen vorgegangen werden:

Im Projekt json2conf auf Unterodner Properties doppelclicken.

Im sich öffnenden Editor links auf Debugging, dann in der mitte im Bereich "Start options" hinter dem vorhandenen Eintrag "..\..\..\smarthome\items\" als 2. Argument den Pfad zum *.knxproj-File eintragen. Wichtig: Der Pfad muss in Anführungsstrichen (") stehen.



Beispiel: Wenn das File Mein Haus.knxproj heißt und im Ordner C:\KNX \Projekte liegt, würde man "C:\KNX\Projekte\Mein Haus.knxproj" eintragen (einschließlich der Anführungsstriche).

Plugin-Support

Grundsätzlich gibt es bei Plugins nichts besonderes zu beachten, solange man alle Plugin-Properties in der Grundform "property": "Wert" notiert. Aus Sicht des Konverters werden Plugin-Properties wie benutzerspezifische Properties behandelt.

Für die Plugins, die ich benutze, habe ich die Plugin-Properties ins JSON-Schema eingearbeitet, hier gibt es also die gleiche Unterstützung wie für die sh.py properties.

Folgende Plugins werden bereits vom JSON-Schema unterstützt (falls noch irgendwelche Properties fehlen, bitte melden):

```
knx_plugin
sqlite_plugin
smartvisu_plugin
network_plugin
smarttv_plugin
fritzbox_plugin
wol_plugin
uzsu_plugin
tla_plugin
autoblind_plugin
xbmc_plugin
speech_plugin
```

Die Liste kann ich gerne erweitern, idealerweise wird sie von denjenigen erweitert, die ein bestimmtes Plugin brauchen und ich übernehme das ins Haupt-Schema.

Zum Hintergrund: Es gibt nicht nur ein Schema, sondern für jedes Plugin ein eigenes Schema - so kann jeder nur die Plugin-Schema aktivieren, die er auch braucht.

Die einzelnen Plugin-Schema werden in der Haupt-Datei ItemSchema.json eingebunden und stehen so zur Verfügung.

AutoBlind-Plugin

Das AutoBlind-Plugin verwendet an einigen Stellen Items, die vollständig vom Plugin interpretiert werden und deren Properties und Unteritems nichts mit dem smarthome Item gemein haben. Aus diesem Grunde habe ich für das AutoBlind-Plugin das Schema erweitert und die speziellen Anforderungen des Plugins eingearbeitet. Damit wird die Verwendung des Plugins komfortabler und die Schema-Verifikation besser, allerdings muss dafür eine

Einschränkung in Kauf genommen werden:

Damit die Schema-Prüfung erkennen kann, dass es sich um den Speziallfall AutoBlind handelt, muss es ein Item mit dem Namen "autoBlind" (exakt so geschrieben) geben. Dieses Item muss das Unteritem "Rules" haben. "Rules" ist dabei das Item, das in der Doku vom AutoBlind als Object-Item definiert wird. "Rule" trägt dann das Property as_plugin, das für den Konverter die beiden Werte "active" und "default" tragen kann. Bei "default" werden diese Items für das default-handling von AutoBlind (siehe doku dort) herangezogen.

Das Item "autoBlind" kann auch ein Unteritem "State" haben, welches die Properties "as_lock_item", "as_suspend_item", "as_suspend_time", "as_suspend_watch", "as_laststate_item_id" und "as_laststate_item_name" tragen kann. Die Nutzung dieser Properties in State wird durch entsprechende code-completion Vorschläge bei der definition von "Rules" unterstützt.

Weiterhin ist die Freiheit bei der Namenswahl (action_name) bei den Konstrukten as_item_(action_name), as_value_(action_name), as_min_(action_name), as_max_(action_name), as_negate_(action_name), as_minage_(action_name) und as_maxage_(action_name). Für alle gilt, der für (action_name) gewählte Name MUSS mit einem Großbuchstaben beginnen. Nur dann wird er mittels des JSON-Schema verifiziert.

Wichtig: Man kann das AutoBlind-Plugin natürlich auch nutzen, ohne sich an die hier beschriebenen Vorgaben zu halten, man verliert dann allerdings die Schema-Prüfung.

Der Vorteil der Schema-Prüfung überwiegt miener Meinung nach die Nachteile, denn der Konverter prüft:

- bei allen as_item_(action_name) referenzen, ob das referenzierte Item existiert
- bei allen as_item_(action_name) werden relative Referenzen ausgewertet und geprüft.
- bei allen as_value_(action_name), as_min_(action_name), as_max_(action_name), as_negate_(action_name), as_minage_(action_name) und as_maxage_(action_name), ob der action_name aus als as_item_(action_name) definiert worden ist.
- bei allen as_... Properties, ob diese im erlaubten Wertebereich liegen.
- bei allen item: und eval: properties werden die items aufgelöst (falls

relativ formuliert) und auf Existenz geprüft.

Dies verhindert sehr viele Tippfehler und ist bei Änderungen der Item-Struktur unglaublich hilfreich.

Schemaerweiterung für benutzerspezifische Properties

Wie oben bereits erwähnt, erlaub sh.py die Benutzung von benutzerspezifischen Properties. Um für diese Properties eine Schema-Unterstützung zu erhalten, kann man diese in ein eigenes Schema eintragen.

Unter smarthome/schema befindet sich die Datei ItemSchemaOwn.json. In dieser Datei stehen 2 Beispiel-Properties, die auf Einfache weise zeigen, wie man wetiere Properties definieren kann. Diese Properties sind derzeit auskommentiert.

Wenn man diese Funktionalität nutzen möchte, muss man die Datei ItemSchemaOwn.json ändern:

Mit Doppelclick auf ItemSchemaOwn.json die Datei im Editor öffnen.

Jetzt in allen Zeilen, die mit // anfangen, eben diese // entfernen.

Zum Test kann man jetzt irgendein items/*.json File aufmachen und dort als Property ***"te*** tippen, dann sollte in der Auswahl test und testEnum erscheinen, das sind die beiden in ItemSchemaOwn.json definierten Properties.

Schemaerweiterung für (noch) nicht unterstützte Plugins

Das Vorgehen ist ähnlich wie bei eigenen Properties. Man kopiert die Datei smarthome/schema/own_plugin.json in eine neue Datei, Namensvorschlag ist

<pluginname>_plugin.json

Diese Datei muss auch ins Hauptschema eingebunden werden. Dazu einen entsprechenden $ref-Eintrag hinzufügen, analog zu den vorhandenen Einträgen, die auf <pluginname>_plugin.json - Dateien verweisen.

Wie ein Plugin-Schema aussieht, kann man in einer der Plugin-Dateien abgucken.